CS4605/Lab 6

George W. Dinolt

September 1, 2004

1 Introduction

The goal of this lab is to illustrate more about the "layering" approach for security models. Another consequence of working on this exercise (I hope) is that you will get a better appreciation of what needs to be proved where. You will note that the order of application controls how things are expanded and as a result whether the proof works or not.

You may want to study the spec as you read the rest of the lab. The spec[1] uses the *pvs* syntax to illustrate how the layers could be built and how the mappings are directly achieved between layers. To achieve the goal, I would like you to prove the final theorem: `fs_is_secure`. If you use the first specification, you can do this in steps by proving the *lemmas* and *tccs* in the specifications as well as the preceding lemmas. The final theorem is a direct result of the previous work.

2 Some Background

2.1 Level 1 - Abstract States

As you know, we have described a "top level" model, the specification works on sequences of abstract things called `states`. There is a notion that some states are *secure* and others are not. The sequences are formed by a state transition function from state to state. If the sequence of states formed this way satisfy:

[1] This is a "hot link" in acrobat and will point you to a file http://www.nps.navy.mil/cs/Dinolt/Courses/AY2004/Summer/CS4605/LABS/lab6/e_sec_theory.pvs in my home directory.You can also find the file on **proof** in the file $/disk1/cisr/Labs/lab6/e\_sec\_theory.pvs$

1. The initial state of the sequence is secure and

2. The transformation rule guarantees that every state in the sequence after the initial one is generated by a transform (or transition) from the previous one

3. Every transform (or transition) from a secure state yields a secure state

then every element of the sequence will be secure.

This structure is described in the `e_sec_theory` specification in the file. The secure *initial state*, *transition is secure* and *sequences are formed by transitions* are all assumptions about the types that are input into the specification.

2.2 Level 2 - Defining states

In this level (`triv_state`) we particularize the notion of a State. We provide a more concrete definition of state, along with this, we define the notions of *secure state* and *transform*. In our simple case, the state consists of 3 entities,

1. An input wire,

2. An output wire and

3. a piece of data.

The interpretation is that the data was received on the input wire and sent on the output wire. Note that we don't care what specific "types" are used on the wires of the data.

We also define a security label set and show how to associate security labels with wires and security labels with states.

These are still "abstract" definitions that are not instantiated, i.e. we still don't have details about the types used for wires, security labels, data. An implementation will have to instantiate these. Note that we are would still like to be able to "prove" that any sequence of the appropriate type has the secure property, but note, that this "state" definition does not include sequences.

We add the notion of sequences and the assumptions on generating them in the specification `triv_system`. It is at this point that we have a general theory that now includes sequences and secure sequences. The security of these sequences is obtained automatically (auto-magically) by including `e_sec_theory` in the specification. This forces us to prove a series of "tccs" about the transitions and security. But these are almost automatic. We need only reference the assumptions about the inputs of `triv_system`.

2.3 Level 3 - Defining an instance of a system

This is the new concept introduced in this specification. In this part we provide an instantiation for a sample system (`sample_system` in the files) that describe how each of the various pieces are instantiated. For example we define the input and output wires to be natural numbers (less than 6) and security labels to be natural numbers 0 and 1. We also construct arbitrary functions that map wires into security labels, etc..

Within the sample system we construct a particular sequence. The goal of this exercise is to prove that this sequence is secure.

3 Lab Details

3.1 Comments on prover commands

You may find the following information useful in completing your proofs.

- "AXIOMS" and "ASSUMPTIONS" can be applied in the same way as lemmas and theorems in "use" and "lemma" statements.
- "rem" stands for the remainder (modulus) function. $rem(6)(1) = 1$ while $rem(6)(17) = 5$. You can find out more about "rem" in the "prelude.pvs" file. An interesting lemma from the prelude that you may want to use (without proof) is `rem_zero` which claims that $rem(n)(0) = 0$ for all natural numbers n.
- You probably want to read up on the "inst" prover command. The syntax is a little strange but you can tell "inst" what to substitute for what and in which sequent or consequent. You use "inst" in quantified expressions.
- You probably want to use the "replace" prover command to do substitution. The command

  $$(replace\ -1\ -2\ rl)$$

  requests the prover to use the equality in sequent -1 replacing the occurrences of the right hand side of the equality with the left hand side in sequent -2.
- the "if-lift" command is useful for managing if statements in the "consequent"

- The "expand" command is often useful. You use expand to expand functions in either sequents or consequents. Note that once a sequence has been "defined," you can expand that as well. Note that $nth(fs, 0)$ expands to $fs(0)$, using `(expand ''nth'')`. If $fs(0)$ is defined to have a value, then one can `(expand ''fs'')` to see what that value is.

- You can find the definition of "every" in the prelude. You can, of course, expand that.

- You may need to generate and prove some `tccs`, as was demonstrated in class.

- You may need to use the "hide" command to help the system instantiate the right things when apply either the "use" or "lemma" commands. Application of these commands may cause the "0^{th}" element of the sequence to be instantiated when you really want the "n^{th} element. The "hide" command can be used to remove a sequent from consideration, leaving the right thing.

3.2 Deliverables

There are two deliverables for this lab.

1. A picture (with discussion if necessary) that shows the layering and how the proofs should work using the layering.

2. The output of the show-proofs-importchain emacs command, issued from the bottom specification of the file. You know you are okay if this shows that the last "theorem" is complete.

Good Luck!